Advances in Engineering Software xxx (xxxx) xxx–xxx

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Research paper

An optimization approach for agent-based computational models of biological development

Pablo Gonzalez-de-Aledo[*,a], Andrey Vladimirov[d], Marco Manca[b], Jerry Baugh[c], Ryo Asai[d], Marcus Kaiser[e,f], Roman Bauer[f,e]

[a] Software Performance Optimization Group, Imperial College London, London, United Kingdom
[b] CERN Openlab, IT Department, CERN, Geneva 1211, Switzerland
[c] Intel Corporation, Santa Clara, CA 95052, United States
[d] Colfax International, 750 Palomar Ave, Sunnyvale, CA 94085, United States
[e] Interdisciplinary Computing and Complex BioSystems Research Group, School of Computing, Newcastle University, Newcastle upon Tyne, United Kingdom
[f] Institute of Neuroscience, Newcastle University, Newcastle upon Tyne, United Kingdom

ARTICLE INFO

Keywords:
Optimizations
Parallel computing
Vectorization
Coprocessor
Performance
Agent-based models
Biological
Simulation

ABSTRACT

Current research in the field of computational biology often involves simulations on high-performance computer clusters. It is crucial that the code of such simulations is efficient and correctly reflects the model specifications.

In this paper, we present an optimization strategy for agent-based simulations of biological dynamics using Intel Xeon Phi coprocessors, demonstrated by a prize-winning entry of the "Intel Modern Code Developer Challenge" competition. These optimizations allow simulating various biological mechanisms, in particular the simulation of millions of cells, their proliferation, movements and interactions in 3D space. Overall, our results demonstrate a powerful approach to implement and conduct very detailed and large-scale computational simulations for biological research. We also highlight the main difficulties faced when developing such optimizations, in particular the assessment of the simulation accuracy, the dependencies between different optimization techniques and counter-intuitive effects in the speed of the optimized solution. The overall speedup of 595 × shows a good parallel scalability.

1. Introduction

With the recent improvements in computing performance, it has become possible to conduct very detailed and large-scale computational simulations for biological research (e.g. [1–7]). However, the efficient use of computing resources remains a major topic in computational biology.

Agent-based models are a powerful computational approach for research on many topics [8]. These models often involve large numbers of interacting agents, and so are usually very demanding from a computational resource point of view. Along these lines, a number of studies have used high-performance computing for agent-based computer simulations. For example, Deissenberg et al. model the European economy by incorporating millions of agents [9]. In biological simulations, agent-based models usually are multi-scale, including interactions between intracellular, extracellular and cell behavioral dynamics in space. The question of how to implement models for the efficient simulation of such computation-intense biological problems is an important research topic in computational biology (e.g. [10–12]), and the application of modern code development approaches has big potentials to advance this field in various biological scenarios. Along those lines, we here focus on an examplary scenario to maximize the efficacy of multi-core simulations relevant to developmental biology.

In particular, we address general optimization techniques for spatial, agent-based simulations in developmental biology. These simulations comprise millions of cells, the interaction among these, as well as their movements in 3D space, and a computational load that changes during simulation. Our study involves the application of parallel coprocessors in high-performance supercomputing. The optimization of code for such hardware is in its infancy, and recent studies demonstrate impressive improvements for scientific simulations [13,14]. However, it remains underinvestigated how biological agent-based simulations that incorporate multiple interacting scales can benefit from such hardware.

Although various programming interfaces and operating systems ease the transition from sequential to multi-threaded parallel code,

* Corresponding author.
E-mail addresses: pgonzal5@ic.uc.uk, pabloga@teisa.unican.es (P. Gonzalez-de-Aledo), andrey@colfax-intl.com (A. Vladimirov), marco.manca@cern.ch (M. Manca), jerry.r.baugh@intel.com (J. Baugh), ryo@colfax-intl.com (R. Asai), marcus.kaiser@ncl.ac.uk (M. Kaiser), roman.bauer@ncl.ac.uk (R. Bauer).

https://doi.org/10.1016/j.advengsoft.2018.03.010
Received 16 July 2017; Received in revised form 2 March 2018; Accepted 20 March 2018

fully-automated parallelization is very difficult to archive due to the lack of adaptation of compilers and profilers for the underlying hardware structure and the data that feeds the program at hand. Therefore, there is still a huge variation in performance due to different programming styles across functionally equivalent versions of the same code.

In order to explore and assess the performance of current optimization techniques for parallelized scientific software, Intel(R) organized in 2015 the Intel Modern Code Developer Challenge. The primary goal of this challenge is to expose students and researchers to the field of parallel computation with the Xeon Phi platform and the Intel compiler, by teaching modern parallelization techniques that not only increase the performance of a given code on a well-known and established platform, but also keep the code portable for future generations of the same platform. Not less important, the challenge is based on a common language and encourages the discussion of new programming techniques for parallel computation. Moreover, it serves as a showcase for an automated and semi-automated parallelization strategy, using the Intel Parallel compiler.

The Intel Modern Code Developer Challenge took place in October 2015 and comprised the optimization of code for simulating the formation of biological tissue in the early stages of brain development. This code allows the simulation of millions of neural progenitor cells that interact with each other biochemically in 3D space. In particular, it involves a number of fundamental processes during the formation of the brain; namely cell proliferation, migration, and secretion as well as detection of diffusable substances and their concentration gradients. Understanding how these key mechanisms of brain tissue development play out, by taking into account genetic factors in a spatially and temporally dependent way, is crucial for the identification of the causes and potential treatments for neurodevelopmental diseases, such as epilepsy, autism and schizophrenia [15–17]. This code has been developed in the context of a collaboration between CERN openlab and Newcastle university, called the BioDynaMo project [18]. The challenge was accepted by over 17.000 students representing more than 130 universities across 19 countries. The criterion for evaluating the entries was based on the optimized execution time as well as the correctness of the final implementation. The former was measured in the same cluster that the students used to test their optimizations, which is described in Section 3. The latter was ensured by the inclusion of two functions in the code that check the final energy of the cellular clusters as well as by manual inspection by three expert judges from Intel.

As part of the facilities offered to the students, Colfax provided remote access to Intel Xeon processor and Xeon Phi coprocessor-based clusters (whose architectural details are described in Section 3). Students were provided free copies of the Intel Parallel Studio XE Cluster Edition and over 20 h of instructional material (that was used by over 1.000 participants).

In this work, we present the results obtained from one of the prize-winning submissions of the challenge (2nd place), aiming at the optimization of the aforementioned neuroscientific code. Importantly, this particular simulation example at hand comprises processes relevant for many problems in computational biology, because they involve intracellular processes as well as intercellular communication via physical mechanisms. Moreover, the code yields remarkable performance also on architectures other than used in the Intel Modern Code Developer Challenge.

From a computational perspective, one distinguishing feature of developmental models is the dynamic nature of the computational load: the developing brain comprises only a small number of cells at the beginning, but subsequently the system size increases exponentially. Hence, the allocated computing resources meet temporally changing requirements during simulation.

Overall, the performance and correctness of optimized code are paramount factors for the explanatory power and scientific practicality of computer simulations of biological dynamics. The optimized code described in this manuscript, as well as the code of the first and third winning entries are provided as supplementary material.[1] Due to the fact that there are many possible interleaved ways of optimizing this non-trivial code, a detailed comparison between the three versions is out of the scope of this work. The third winner of the competition also provides an explanation of the optimization techniques that he employed in [19], and a substantial overlap exists between the techniques described in his solution and the ones described in this manuscript.

The main contributions of this paper can be summarized as:

- We present a highly parallel implementation of a computer simulation that involves millions of agents interacting in a 3D environment.
- We study the simulation of biological development using various multi-core platforms.
- We explain a general approach to transform sequential code of biological dynamics to run on modern, highly parallel architectures such as the Intel Knights Landing, Broadwell, Sandy-Bridge and AMD Opteron.
- We present the techniques that enabled us to obtain almost 600 × speed-up over the mentioned platforms and simulations.
- The manuscript exemplifies the innovative use of computational strategies and numerical algorithms for large-scale biological problems.

2. Initial software architecture

The initial architecture of the code can be seen in Fig. 1. The code can be partitioned into two phases. Initially, a single precursor cell is placed in the middle of a 3D space.

2.1. Proliferation phase

In the first phase of the simulation, cells move randomly and divide until the final number of cells is reached. After each cell division, the daughter cells each adopt one of two possible cell types, hence giving rise to cell differentiation. This simulation is performed using the functions *produceSubstances* and *cellMovementAndDuplication*. In the function *produceSubtances*, one of the two substances (a or b) is produced depending on the cell type (+ 1 or − 1). Hence, each cell secretes only one of two possible substances, which is determined by their cell type. Cells of type + 1 secrete substance a, and cells of type − 1 secrete substance b. The dynamics of these two substance concentrations are described by the following partial differential equations:

$$\frac{\partial a}{\partial t} = p - \mu a + D_a \frac{\partial^2 a}{\partial^2 x} \tag{1}$$

$$\frac{\partial b}{\partial t} = p - \mu b + D_b \frac{\partial^2 b}{\partial^2 x}, \tag{2}$$

with the basal production constant $p = 0.1$. The prespecified diffusion constant D and decay constant μ are identical for both substances.

In the functions *runDiffusionStep* and *runDecayStep*, the diffusion and decay of the two cellularly secreted substances are numerically simulated. These functions read and write into the *Conc* 3-dimensional matrix, which stores the concentrations of both substances within a spatial grid.

At the end of the iterative step in the proliferation phase, the function *cellMovementAndDuplication* updates the arrays *posAll, pathTraveled, typesAll*. The array *pathTraveled* includes the overall length of the path that each cell travels. If this value exceeds a given threshold parameter T_{path}, the cell divides and the value is reset.

[1] Also available at https://www.github.com/pablo-aledo/intel-modern-code-challenge.

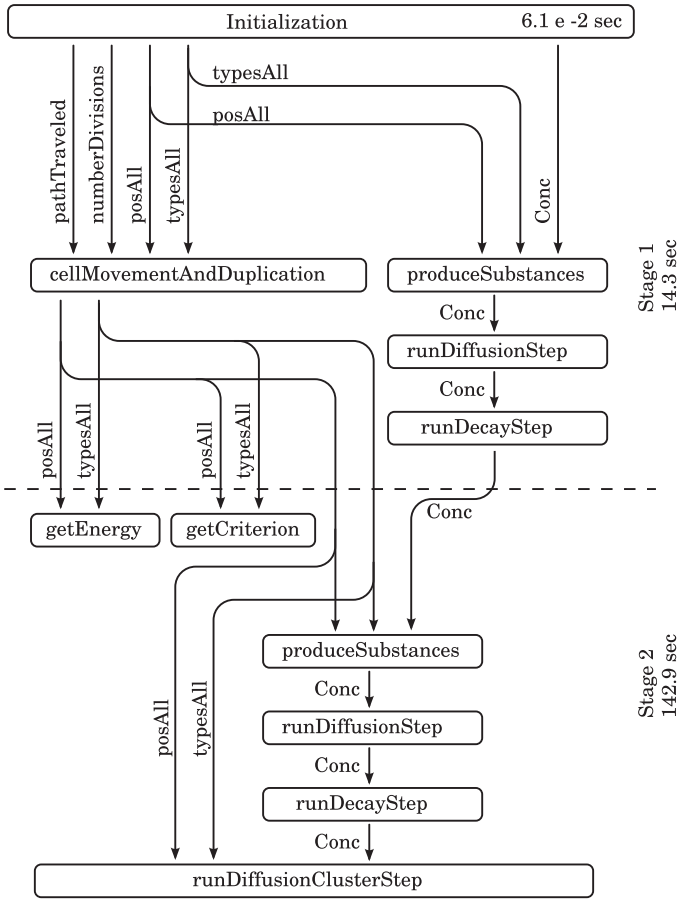


Fig. 1. Data flow of the initial implementation. Times correspond to the parameters described in Table 1 (left). Simulations in the unoptimized version based on the parameter set on the right of Table 1 did not terminate within several hours, and so no time measurements can be given in this scenario.

Importantly, this value is a continuously changing property of individual cells, and so in principle could indicate any intracellular feature such as gene expression or similar. Hence, the function of *pathTraveled* can be understood as keeping track of a continuously changing internal cell state. Analogously, *numberDivisions* keeps track of the number of divisions a cell has undergone in the past, i.e. this variable is updated after each cell division and conveyed to the daughters. Once a cell has reached a given number of maximum divisions (the division threshold model parameter T_{Div}), it stops dividing. The cell division phase ends when all cells have reached this final number of cell divisions.

The array *posAll* stores the cells' 3D locations. Finally, the array *typesAll* indicates the cell types (+ 1 and − 1), and so enables the model to simulate cell differentiation, which is a crucial process of biological development. These arrays are subsequently also used in the second phase of the simulation.

2.2. Cluster formation phase

In the second phase of the program, self-organized formation of cell clusters is simulated (see Figs. 13 and 14). This clustering is based on the movement of cells along gradients of extracellular substances, which is a well known ability of eukaryotic cells [20]. Moreover, during brain development, neuronal connections are also guided by extracellular substance gradients [21]. Hence, in addition to the previous secretion of substances by cells, this phase simulates also the movement of the cells with respect to these substances.

Cells are attracted by and move along the gradient of the substance associated with the respective cell type (i.e. + 1 and − 1 type cells are attracted to substance *a* or *b*, respectively), but move away from gradients of the opposite substance type. This process finally leads to cluster formation, with clusters comprising cells of the same type. We chose to simulate the formation of cell clusters/aggregates because it allows for easy quantification and also visual detection of such cell behavior. During this second phase, "*T*" time steps are simulated, and at each time step *produceSubstances, runDiffusionStep, runDecayStep* and *runDiffusionClusterStep* are run sequentially.

2.3. Model quantities

We set the diffusion and decay parameters D and μ to 0.3 and 0.1, respectively. The speed of cells, indicating a scalar factor by which the (local) gradient direction of the extracellular substance is multiplied, was set to 0.01, T_{path} to 2.0 and the side length of the cube to 5.0. This parameter set allows for clearly visible cluster formation, but only serves as a proof of principle, and other values would be possible (as well as biologically plausible in the given spatial domain) too.

Overall, the following structures are kept updated:

- *pathTraveled*: Array keeping track of the length of the path traveled until cell divides.
- *numberDivisions*: Array keeping track of the number of divisions a cell has undergone.
- *PosAll*: Maintains the position of each cell in the space to be simulated.
- *TypesAll*: Array specifying cell type
- *CurrMov*: Maintains the cell movements in the last time step.
- *Conc*: 3D concentration matrix keeping the concentration of cells in the space to be simulated.

The simulation is confined to a 3D unit cube. The diffusion is computed based on a regular grid comprising $L \times L \times L$ voxels. At the end of the simulation, measurements for the overall energy and a criterion indicating if the clustering has taken place or not are computed (Fig. 4). These measurements quantify the spatial distances between cells, taking into account their cell types. They allow to heuristically indicate whether cells of the same cell type tend to cluster together or not. This is done using the functions *getEnergy* and *getCriterion*, which require the *posAll* array (for information on cell locations) and *typesAll* array (for information on cell types) as their inputs.

3. Hardware architecture

During the contest, the performance of the application was measured on an Intel Xeon Phi 7120P coprocessor. This device, based on the Many Integrated Core (MIC) architecture, comprises 61 cores clocked at 1.238 GHz with 4 in-order hardware threads per core. Each core has a vector processing unit (VPU) with support for Single Instruction Multiple Data (SIMD) operations on 512-bit vectors of floating-point and integer numbers. Cores have symmetric access to a total of 16 GB of memory based on the GDDR5 technology. This memory is cached with 512 KiB of Level 2 cache and 32 KiB of Level 1 data cache per core. Caches belonging to all cores form an aggregate coherent cache with a distributed tag directory interconnected by a high-speed ring bus. The coprocessor functions as a PCI Express add-in card, i.e., it is installed on a host system based on a traditional CPU, with which the coprocessor does not share the memory address space. Nevertheless, once booted, they may be used as stand-alone compute nodes with their own filesystem, networking and memory [22].

The application was executed on the coprocessor in the native mode, i.e., it was compiled to run directly on the MIC architecture, without involving the host processor or memory. The executable, runtime libraries used by it, and the input data file were placed into the virtual filesystem of the coprocessor. When the executable was

launched, it utilized the entire coprocessor, with no other computational workloads running on it.

The code was compiled with Intel C++ compiler version 16.0.0.109. The software stack for Intel Xeon Phi coprocessors was MPSS 3.5.2. The operating system on the host system, CentOS 6.2, used the Linux kernel version 2.6.32-220.

The computing system used for the contest had 8 coprocessors described above installed in two CXP8600 servers produced by Colfax International. Each server, powered by Intel Xeon E5-2697 V2 processors, with 128 GB of DDR3 RAM, housed 4 coprocessors. With multiple users submitting jobs (i.e., runs of the cell clustering application) to the execution queue, one of the available coprocessors was assigned to each job, with job order depending on a usage-based fair share policy. Job scheduler Maui 3.3 was used to choose the next job for execution and calculate the fair share weights. Resource management tool Torque 2.5.7 was used for maintaining the queue and controlling the status of the compute nodes. Because Torque 2.5.7 does not have support for native execution on Intel Xeon Phi coprocessors, the environment of the coprocessors was virtualized, i.e. each server hosted 4 virtual machines acting as compute nodes, with one Intel Xeon Phi coprocessor per virtual machine. Jobs were submitted to Torque as normal CPU-based calculations, but they were forwarded to the respective coprocessor by means of a wrapper script, with staging and cleanup managed by custom prologue and epilogue scripts in Torque. Special care in Linux configuration was taken to ensure that (i) each coprocessor is assigned exclusively to exactly one job, and (ii) contestants do not have ways to override or circumvent the queue.

Despite the complexity of the system used for the contest, results presented here may be reproduced in systems containing at least one Intel Xeon Phi coprocessor with 16 GB of onboard memory. Because of the native programming model used in the calculation, performance results do not depend on the system configuration on the host system.

4. Code optimization

In the following sections, the optimizations conducted to increase code performance of the above described platform are explained. Later on, these optimizations are assessed on other platforms too.

Optimizations are grouped based on the main goal to achieve; in “parallel optimizations”, the main target is to increase the parallelism of the code (i.e. the number of operations that can be conducted at the same time). In “sequential optimization” we focus on single cores and try to reduce the associated computational load; finally in “memory optimizations” cache locality is exploited to provide high-bandwidth and low-latency access to the data.

4.1. Sequential optimization

In this section, general optimization techniques are described. These optimizations do not rely on parallel architectures or on special memory layouts.

4.1.1. Loop transformations

The benefit of loop transformations is twofold; on the one hand, due to limitations in inter-procedural and dependency analysis, compilers do not usually do a good job at detecting parallelization opportunities when multiple loops are involved. On the other hand, loop transformations generally increase temporal and spatial memory locality. In addition to the fact that we have fewer loops in subsequent transformations, and that they are simpler, we also require less fine-tuning of the optimizations, which renders loop-transformation optimizations good candidates to be applied in the first steps of the overall optimization process.

When a loop is vectorized by the Intel Compiler and transformed into a “vectorized” loop, several iterations of the loop are transformed into parallel SIMD operations. Those operations are more efficient when the data that is loaded from memory is aligned to a multiple of the register width. To implement this alignment, the compiler splits the loop into a sequential (or masked vector) execution (the “peeled” part) and a parallel one (the vectorized loop), so the vectorized part can operate the data optimally. Fig. 2 shows how this information is presented in the optimization report presented by the Intel compiler, so suboptimal optimizations can be detected and corrected.

```
Report from: Vector optimizations [vec]
============================================
LOOP BEGIN at cell_clustering.cpp(337,3)
remark #15300: LOOP WAS VECTORIZED
LOOP END
============================================
LOOP BEGIN at cell_clustering.cpp(176,5)
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
LOOP END
============================================
LOOP BEGIN at cell_clustering.cpp(241,7)
<Peeled loop for vectorization>
remark #15301: PEEL LOOP WAS VECTORIZED
LOOP END
============================================
LOOP BEGIN at cell_clustering.cpp(241,7)
remark #15301: SIMD LOOP WAS VECTORIZED
LOOP END
============================================
LOOP BEGIN at cell_clustering.cpp(241,7)
<Remainder loop for vectorization>
remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
============================================
LOOP BEGIN at cell_clustering.cpp(610,6)
<Distributed chunk1>
remark #15344: loop was not vectorized: vector
     dependence prevents vectorization. First
    dependence is shown below. Use level 5
    report for details
remark #15346: vector dependence: assumed
    OUTPUT dependence between  line 611 and
    line 611
LOOP END
```

Fig. 2. Generated report showing information about optimization results. Examples of successful and unsuccessful optimizations are presented. Optimization reports present the internal information to the developers so they can modify the source code to ease the compiler transformations, enabling a semi-automatic vectorization approach.

Some examples of loop transformations are:

- Loop coalescing: As shown in Algorithm 1, the transformation consists in identifying loops that operate over the same input domain, but produces results in different output variables. That enables joining those loops, and exposes more code to parallel computation.
- Loop splitting: On the other hand, if we can identify loops that operate over different variables that do not share any dependencies, it is useful to split them and include the `pragma` directive to enable task-level parallelism as shown in Algorithm 2.

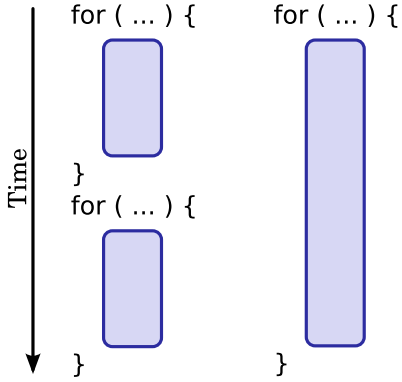
4.1.2. Optimizations and trade-off between precision and speed

Although the applicability of these optimizations depends entirely on the problem at hand, and no already-established well-known techniques exist here, allowing small errors in the results can generally produce drastic improvement in execution times. Selecting approximate algorithms instead of deterministic ones, as well as modifying already coded algorithms to allow for asymptotically correct answers can reduce execution times considerably if we are willing to sacrifice

Algorithm 1
Example of loop join. A loop inside the function *runDiffusionClusterStep* is joined with another that is executed after leaving the function. This loop transformation is useful when the bodies of the loops operate over the same data, because more code is exposed to parallel execution.

```
procedure RUNDIFFUSIONCLUSTERSTEP(Conc,...)
    ...
    for all cell ∈ C do                    ▷ Iterate over all the cells
        ...                                ▷ Run diffusion code for cell
    end for
end procedure
procedure MAIN
    runDiffusionClusterStep(Conc, ...      ▷ Call to runDiffusionClusterStep. This function contains a loop inside that executes the diffusion of every cell.
    for all cell ∈ C do                    ▷ Iterate over all the cells
        ...                                ▷ Update position of cell
    end for
end procedure
```

```
procedure RUNDIFFUSIONCLUSTERSTEP(Conc,...)
    ...
    for all cell ∈ C do                    ▷ Iterate over all the cells
        ... ▷ Run diffusion code and update the position of cell
    end for
end procedure
```



correctness to a certain extent. However, this kind of optimization usually requires expert knowledge in the problem domain, and introduces the possibility of producing incorrect results even when this probability can become arbitrarily small. Some instances of these optimizations include down-sampling of the inputs of the problem, changing the algorithms used in the program, or using mathematical libraries that allow for this kind of trade-off. In the program at a hand, we use the Intel Math Library (R), with model "fast" and precision "low" (the compiler can be instructed to apply these settings with flags `-fp-model fast=2 -fimf-precision=low`).

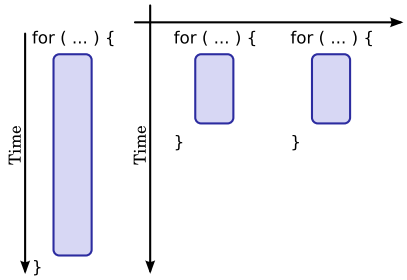
4.1.3. Fixed point operations

Another, more extensive example of optimizations that can sacrifice precision is the transformation of floating-point operations to fixed-point. As already mentioned, due to internal details of the pipeline and architecture, floating point operations can take notably longer times to execute than its fixed-point counterparts. The use of integer datatypes instead of floating point can therefore reduce the computational effort, and also enables to choose the bit-width that is used to represent each variable, therefore introducing another degree of freedom in the trade-

Algorithm 2
Example of loop split. This loop transformation is especially useful when the body of the loops operates over disjoint data, since both loops can then be executed in parallel.

```
procedure PRODUCESUBSTANCES(...)
    for all cell ∈ C do                    ▷ Iterate over all the cells
        ...
    end for
    ...
    for i1 = 1; i1 < L-1; i1++ do
        for i2 = 1; i2 < L-1; i2++ do
            for i3 = 1; i3 < L-1; i3++ do
                ...
            end for
        end for
    end for
end procedure
```

```
procedure PRODUCESUBSTANCES_SECTION1(...)
    for all cell ∈ C do                    ▷ Iterate over all the cells
        ...
    end for
end procedure
procedure PRODUCESUBSTANCES_SECTION2(...)
    for i1 = 1; i1 < L-1; i1++ do
        for i2 = 1; i2 < L-1; i2++ do
            for i3 = 1; i3 < L-1; i3++ do
                ...
            end for
        end for
    end for
end procedure
procedure MAIN(...)
    #pragma omp sections                   ▷ Execute both sections in parallel
    produceSubstances_section1(...
    produceSubstances_section2(...
end procedure
```



off between performance and error.

By their very nature, biological processes have to exhibit flexibility and robustness, hence any representations (e.g. of intracellular states) should not depend on very high precision. Along those lines, the coefficient of variation of the expression of specific genes of nearby cells has been estimated at approximately 8% [23]. Given that the optimized code has a precision of 32 bits, no compromising effects on the scientific value of such biological simulations are entailed. Moreover, comparative simulations yielded good agreement in terms of the final cell distributions within clusters, as captured by the measured "energy" (Fig. 3) and the clustering behavior (Fig. 4). This energy measure takes into account the cell types and all cell-pair distances, quantifying the

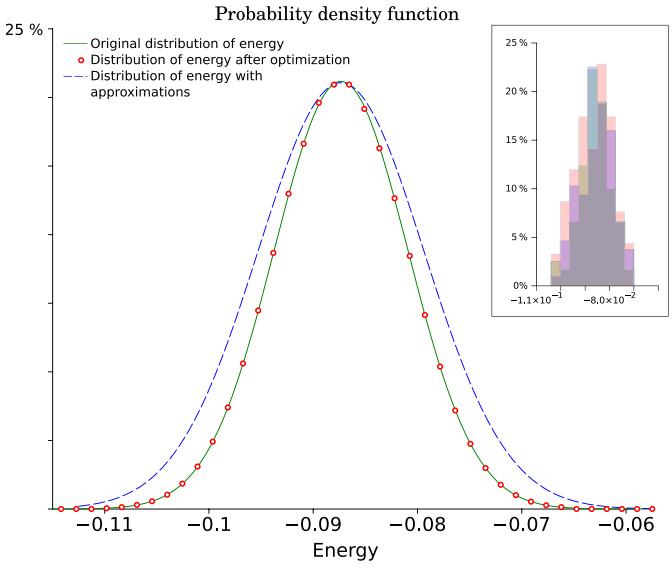


Fig. 3. Distributions of energy after simulation. 100 simulations of the 'small' test case were conducted in the non-optimized (green, solid line), optimized (red circles) and optimized with less precision (blue, dashed line) scenarios. Reassuringly, the loss in precision has minimal impact on the standard deviation of the energy (the blue distribution is slightly wider than the red and green distributions), demonstrating good viability of this optimization step. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

D \ *mu*	0.6	0.65	0.7	0.75	0.8	0.85	0.9
0.15	×	×	/				
0.2	×	×	×	×			
0.25	×	×	×	×	×		
0.3	×	×	×	×	×		
0.35	×	×	×	×	/		
0.4	×	×	×	×	×	×	
0.45	×	×	×	×	×	×	

Fig. 4. Comparison of clustering in an interesting region of the parameter space, showing a good correspondence between the optimized and non-optimized versions of the code. /: optimized version shows clustering behavior, \: non-optimized version shows clustering behavior. For an explanation of the clustering condition, see Section 4.1.3.

tendency for cells of the same type to cluster with one another, while cells of the opposite type are avoided. The implementation of this measure can be found in the code included in the supplemental material. Moreover, a function *getCriterion* was implemented, which returns 0 if the cell locations within a central subvolume of the total system, comprising approximately N cells, are arranged as clusters, and 1 otherwise. To classify a simulation setting as clustered and non-clustered, respectively, we used the following heuristic:

- If the central subvolume comprises less than 1/4 of all cells, the clustering criterion is not fulfilled.
- If more than 1/10 of all cells within a close distance are of the opposite type, this indicates bad clustering.
- If clusters are not large enough, i.e. if cells have less than 100 cells of the same type located nearby, this also indicates bad clustering.

4.1.4. Function inlining

To implement function calling in a standard and organized way, compilers generally introduce special operations. These operations deal with the stack to pass parameters and receive the results. This does not represent a significant time penalty if the function is rarely called because the effect of the stack is negligible in comparison with the computations that are performed inside the function. However, for functions that are called very frequently and whose body is relatively small, it can be more beneficial to introduce the function body instead of the call (or use the 'inline' keyword). This makes the code larger, and therefore more cache misses will occur in the instruction cache. Therefore, this optimization introduces a balance that has to be considered and is not recommended to be applied in the first stages of the optimization flow. Some candidate functions to be inlined in the code are 'getNorm', 'RandomFloatPos' and 'getL2Distance'.

4.1.5. Optimizations targeted to reduce the number of executed operations

The first set of optimizations applied to the code are aiming at reducing the number of executed operations. Common instances of this kind of optimizations are factorizing common sub-terms that are used multiple times. The compiler is generally unable to detect these optimization opportunities due to inter-procedural analysis. A good example of such limitations can be found in the pair of functions *getCriterion* and *getEnergy*, whose computations are very similar, but due to the fact that the commonalities are among two different functions, the code cannot be reused. Moving the common code outside both functions and passing the required values to both functions produces, in this case, a speed-up of 1.25.

Another example of this transformation can be seen in Algorithm 3. In this transformation, a sub-expression that is common to several instructions inside the loop is identified and extracted out of the loop to avoid unnecessary computations.

4.1.6. Optimizations to remove expensive operations

Due to complex datapaths and the internals of modern architectures, different operations can take a different number of cycles to finish, so a good optimization technique is to transform expensive operations by mathematically equivalent ones that are much cheaper to compute, especially if those operations take place in loops so that they are executed multiple times. Common examples of this transformation are focused on multiplications and divisions since these are usually the most expensive ones in modern pipelines. Compilers are usually able to transform multiplications of variables with integer constants by equivalent operations that include shifting and adding (i.e. a multiplication of the variable a with the integer 3 can be expressed as $a + (a << 1)$ where the operator $<<$ means shifting the binary representation of a one bit to the left). However, in some occasions, the compiler does not have enough information to infer an equivalent expression. Fig. 5 shows one of these examples.

4.2. Parallel optimizations

In this section, optimizations that are mainly focused on exploiting the high degree of parallelism available in the platform are described.

4.2.1. High-level task parallelism

Task dependencies of the computation described in the introduction can be seen in Fig. 1. As can be seen, several computations can already be performed in parallel to the initial implementation of the source. However, the optimizations achieved by a simple parallelization like this are sub-optimal. To increase the parallelism in the whole program we need some architectural transformations.

- Double buffering has been used in the array *Conc*. This is a common technique when implementing stencils in the code, and more details can be seen in references [24,25]. In the original implementation, the output of the stencil is applied to the same array that constitutes the input. To keep the input data and not overwrite it with the output, however, a copy is needed, so a new array that is allocated inside the function is used as a temporary result buffer. This

```
for (i1 = 0; i1 < L; i1++) do
    for (i2 = 0; i2 < L; i2++) do
        for (i3 = 0; i3 < L; i3++) do
            Conc[0][i1][i2][i3] += ... * D*0.166*(1-mu);
            Conc[0][i1][i2][i3] += ... * D*0.166*(1-mu);
        end for
    end for
end for
```

```
float factor = D*0.166*(1-mu);
for (i1 = 0; i1 < L; i1++) do
    for (i2 = 0; i2 < L; i2++) do
        for (i3 = 0; i3 < L; i3++) do
            Conc[0][i1][i2][i3] = ... * factor;
        end for
    end for
end for
```

Algorithm 3. A common sub-expression is detected inside a nested loop. The sub-expression does not change its value on different iterations, but the compiler can not be aware of this fact because of complex array processing (it cannot ensure that the variables 'D' or 'mu' are not pointed to by 'Conc' during some of the writings). To avoid the repeated computation, the sub-expression is renamed as "factor" and extracted out of the loop.

```
gradSub1[0] = (conc(0,xUp,i2,i3)-conc(0,xDown,
    i2,i3))*L/(xUp-xDown);
```

```
float factorx = (xUp-xDown)==2?LOver2:L;
gradSub1[0] = (conc(0,xUp,i2,i3)-conc(0,xDown,
    i2,i3))*factorx;
```

Fig. 5. In the presented code, we know that due to the nature of the computations that are performed before these operations, the only possible values of $xUp - xDown$ are 1 or 2, and we also know that the value of L does not change from one iteration to the next one. This enables us to precompute the possible values of the division $L/(xUp - xDown)$ and use them instead of computing the division each time.

introduces the overhead of creating the buffer and copying the input data to it, and so it is generally better to use a second buffer and swap both arrays on each iteration. If the former buffer is used as input in iteration n, the latter is used as output, so there is no overwrite. After executing every iteration, both buffers are swapped so the net effect is the same. This requires twice as much memory, but removes the need to copy the buffer in the *produceSubstances* function. Even though this transformation requires the modification of all functions that access the buffer to take into account the new memory layout, Fig. 12 shows that this optimization is one of the most effective when applied to the code.

- High-level parallelism has been implemented for the functions that do not have dependencies among them. Such dependencies exist between *produceSubstances* with *cellMovementAndDuplication* and *produceSubstances* with *runDiffusionClusterStep*.

4.2.2. Vectorization

Besides 'task-level parallelism', on a lower level of detail, using vector instructions inside functions can increase substantially the performance of the code (Fig. 6). Thanks to the auto-vectorization features of the Intel compiler, we can keep a high-level abstraction of the functionality while still using low-level features of the architecture such as wide vector words. Vectorization has been used extensively in the code, but the function in which its effect is most noticeable is *cellMovementAndDuplication*. In this function, a random movement is applied to each cell, and depending on the number of divisions that the cell has undertaken, it might or might not divide to create new cells. Due to the fact that the cell positions are stored in a contiguous array, this function is a good candidate for vectorization. In a first optimization (shown in Algorithm 4), the loop is vectorized in such a way that at each iteration, a random vector of size 3 times the number of cells is created and added to the position of all cells. Given that the initialization of the random vector causes a bottleneck, however, a second optimization has been applied in which the random vector is only initialized once at the beginning of the execution, adding a slack at the end of the vector to accommodate a few more elements than the ones strictly needed. During execution, a random number is chosen at every iteration, and this number is used as an offset to start adding random positions to the cells. This keeps the benefits of vectorization since both arrays are accessed in contiguous order, and also improves the execution time because there is no need to initialize the random vector on each iteration.

4.2.3. Pointer aliasing

In order to implement automatic vectorization, the Intel compiler performs a static analysis that ensures the correctness of the vectorized code in relation with the original one (pointer aliasing analysis). This analysis is particularly difficult when multiple functions are involved because inter-procedural analysis is needed, and the compiler does not know how each function might be possibly called. To ease the work of the compiler, the developer has to introduce "contracts" or "promises" that guide the analysis indicating that there are no pointer aliases for the possible calls to specific functions in the code. There are many

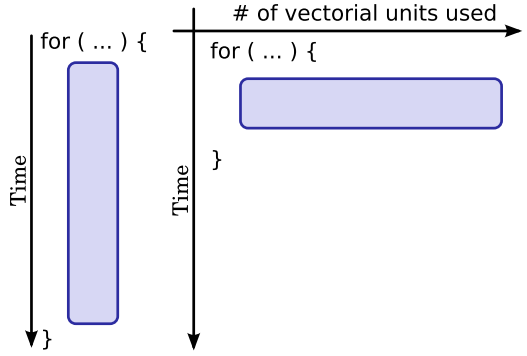


Fig. 6. Example of loop vectorization. This transformation is particularly useful when the loop operates over contiguous elements in the memory layout.

different forms to convey this information to the compiler. One example is presented in Fig. 7; in this example, we rely on the "restrict" keyword because of its ease of use. Using the keyword "restrict" in front of a parameter in a function declaration instructs the compiler that the array to which this parameter points is only accessed through that pointer, and therefore no aliasing is possible.

4.2.4. Parallel reduction

Several loops in the original code are used sequentially in conjunction with a global scalar available to all iterations of the loop to compute sums of different terms or to count the number of times a certain event happens. The way in which the loops are implemented introduces data dependencies caused by reads and writes in the global object, and therefore a parallelization is impossible. To remove these dependencies, a common technique in parallel programming is to introduce an array or a matrix that stores the result of each iteration individually. This usually enables executing most of the work in parallel, since each thread writes in its own private variable and therefore there are no collisions. A final sequential step is then applied to compute the serial part, in which no parallelization is possible. This technique called "parallel reduction" has been used in the functions *cellMovementAndDuplication* to count the number of divisions of each cell, and in *getEnergy* and *getCriterion* to compute the variables *intraClusterEnergy, extraClusterEnergy, nrSmallDist, nrClose, diffTypeClose* and *sameTypeClose*.

4.2.5. Loop scheduling and thread affinity

Once the other optimizations related to loops have been applied, further extra-performance can be obtained by fixing a concrete scheduling policy for the most demanding loops. Choosing a good policy consists in obtaining a good trade-off between a fixed scheduling in which time can be wasted due to load imbalance, or a dynamic policy in which time can be wasted when planning the tasks at every iteration. Usually, the designer of the original code knows which loops are balanced and which ones are not, but trying different scheduling criteria can help to determine which policy should be applied to each loop. In the case of our program, the most demanding loops are the ones in *produceSubstances*, and the best performing scheduler for both of them is "guided" scheduling, which implements a trade-off between the two cases considered previously.

Similarly, the Intel OpenMP runtime library enables to fine-tune the affinity of threads to physical processing units in the platform. Thread affinity restricts execution of certain threads to a subset of the physical processing units in the computational platform. The considered affinity configurations are *none* (do not bind OpenMP threads to thread contexts), *compact* (assign threads to particular CPUs in the platform) and *scatter* (try to distribute the threads as evenly as possible across the entire system). The highest speedup was obtained when *compact* affinity was used. To report the speedup figures of graph in Fig. 15(b), the affinity has been set to *compact, 1* to map each thread to a single CPU and obtain stable results.

4.3. Memory optimizations

As noticed in [26–28], the memory layout usually constrains algorithms more than the computational speed. Even when the simulation code is optimized for the maximum amount of operations in parallel, and the work given to each core is reduced and the load is balanced, without memory optimizations the improvement will still be limited by the memory bandwidth. This bandwidth determines how much data can be provided to the computational cores at the appropriate time. This effect is commonly named "memory wall" [26–28].

4.3.1. Memory allocation, first-touch, and alignment

For allocating data in many-core architectures, the Intel compiler provides libraries that implement the function `mm_malloc`. This function implements a lazy initialization policy that defers the allocation of the memory until a core tries to use it. This gives the system a "hint" to which physical memory region the array should be allocated on and generally produces the beneficial effect of allocating the memory close to the core that uses the data most frequently.

Due to vectorization, arrays need to be of size multiple of 16. To avoid that the Intel compiler introduces "peeling" in loops over arrays that are not multiples of 16, the size has been artificially incremented in these arrays to the closest multiple of 16, and a few extra iterations are performed over these arrays. To do so, the directives shown in Fig. 9 are used in order to allocate a uni-dimensional array from a multi-dimensional array (in this case a 4-dimensional concentration array).

Now that all the arrays are allocated as uni-dimensional vectors, we need special macros to retrieve back the row and column in multi-dimensional arrays. We can do this via pre-processor directives, as shown in Fig. 10.

4.3.2. Optimizations to improve temporal and spatial data locality

To maximize the utilization of cores and avoid hitting the "memory wall", it is important to maximize cache utilization. To this end, two techniques have been used in the implementation of the most demanding functions: loop tiling and cache-oblivious algorithms. In the former technique, the input data is partitioned into "tiles", and loops that operate over that data are transformed in such a way that they scan the data first with a stride that is equal to the tile size, and then a second loop operates "inside" the tile. This has the advantage that locality is increased, and in general, more accesses are kept in the low levels of the memory hierarchy. To further increase the effect, cache-oblivious techniques [24] have been used as well.

5. Results

In this section, the results of the aforementioned optimizations are discussed. To this end, two graphs are presented. The first one (Fig. 11) illustrates the order in which optimizations have been performed, as well as the relative effect that each one of them has produced. This figure shows the execution time vs. the version committed in the version-control repository that this project is associated with, and it is indicated to which category the optimization technique belongs to.

In Fig. 12, the optimizations performed in the code are classified in terms of "optimization target", as explained before. The relative effect of each optimization can be seen in this graph. We can see in this chart that the most important optimizations for the problem at hand involve loop transformations and memory layout techniques.

In these figures, two examples are shown. The former is called 'small' test case and the latter 'huge' test case. These names summarize different simulation parameters that are presented in Table 1. The purpose of the small set of parameters is to be able to quickly test the effect of optimizations in a manageable test case, while the purpose of

```
for all cell ∈ C do                                   ▷ Initializes the movement and norm of a random displacement for the cell
    float currentCellMovement[3];
    currentCellMovement[0]=RandomFloat()-0.5;
    currentCellMovement[1]=RandomFloat()-0.5;
    currentCellMovement[2]=RandomFloat()-0.5;
    currentNorm = getRNorm(currentCellMovement);
end for
```

```
VSLStreamStatePtr rnStream;                                   ▷ Declaration of the vectorized random number generator
float* RandomFloat_v = (float*)malloc(RandomFloat_Pad*sizeof(float));   ▷ Declaration of the displacement array
float* squares = (float*)malloc(RandomFloat_Pad*sizeof(float));   ▷ Auxiliary vector to store the components squared
float* sqrts = (float*)malloc(RandomFloat_Pad_2*sizeof(float));   ▷ Magnitude of the displacement
vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, RandomFloat_Pad, RandomFloat_v, -0.5f, 0.5f);  ▷ Fill the
displacement array with random values
#pragma simd
for (c=0; c< 3 ∗ n; c++) do                                   ▷ Vectorial computation of the components squared
    squares[c] = RandomFloat_v[c] * RandomFloat_v[c];
end for
#pragma simd
for (c=0; c<n; c++) do                                        ▷ Vectorial computation of the magnitude
    sqrts[c] = 1.0/(squares[3*c] + squares[3*c+1] + squares[3*c+2]);
end for
```

Algorithm 4. Example of vectorization as implemented in *cellMovementAndDuplication*. In this function, the movement of an array of cells is initialized with random three-dimensional vectors. To vectorize the code, the function *RandomFloat* (that returns a *single* random value) has been replaced by the vectorized random number generator vsRngUniform. The computation of the module of each of these three-dimensional vectors can be vectorized as well by using an auxiliary vector to store the components squared. The Intel compiler is able to detect the interlacing of the values stored in the array, and produce an efficient vectorized implementation without the programmer dealing with the low-level details.

```
static void produceSubstances(float* Conc,
    float* posAll, int* typesAll, int L, int n
    , float sideLength, float D, float mu) ...
```

```
static void produceSubstances(float* restrict
    Conc, float* restrict posAll, int*
    restrict typesAll, int L, int n, float
    sideLength, float D, float mu)...
```

```
CFLAGS = ...
```

```
CFLAGS = ... -restrict
```

Fig. 7. Transformations in the source code and in the Intel-gcc compiler to enable pointer-analysis and semi-automatic loop vectorization.

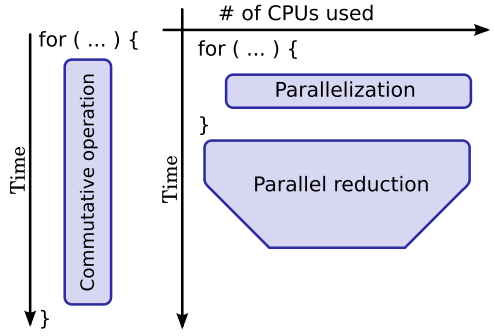


Fig. 8. Example of parallel reduction. This transformation can be applied when the purpose of the loop is to update a variable with the result of a associative operation over the input domain.

```
int array_pad = n+(n%16 == 0? 0:16-n%16);
float* array = (float*) _mm_malloc(array_pad*
    sizeof(float), 64);
```

```
int matrix_pad = m+(m%16 == 0? 0:16-m%16);
#define matrix_access(a,b) matrix[a *
    matrix_pad + b]
float* matrix = (float*) _mm_malloc(n*
    matrix_pad*sizeof(float), 64);
```

Fig. 9. Declarations to access multi-dimensional arrays with the constraints of vectorization.

```
float** Conc;
Conc = new float*[L];
for (i0 = 0; i0 < L; i0++) {
  Conc[i0] = new float[L];
  for (i1 = 0; i1 < L; i1++) {
    Conc[i0][i1] = zeroFloat;
  }
}
```

```
#define Conc(a,b) Conc[ a*Conc_Pad + b]
Conc_Pad = L+(L%16 == 0? 0:16-L%16);
float* Conc = (float*) _mm_malloc(Conc_Pad*
    Conc_Pad*sizeof(float), 64);
```

Fig. 10. Preprocessor directives to access the elements of an n-dimensional array and account for alignment in memory allocation.

the "huge" one is to perform the actual high-performance simulation. As we can see in Figs. 11 and 12, however, the effect of some optimizations is different over the two examples. Also, note that the initial steps of the optimization procedure presented in this work could not be measured for the huge test case due to the lack of time and computational resources, so in Fig. 12, only the results after certain optimizations are accounted for. The improvements obtained and presented in gray in Fig. 12 can be seen as the relative importance of different techniques once the most trivial optimizations (the low hanging fruit) have been applied.

Seeing Figs. 11 and 12, we can sort the most important performed optimizations as: double-buffering, parallelization, tiled or cache-oblivious implementation and approximation. In the case of the huge test case, some of these transformations could not be measured, since the execution at the initial stages was too time-consuming. Because of that, only the first implementation of the sequential code has been simulated with the *huge* test case—the complete simulation took 45 h in the Xeon E5. However, we can see in the examples that these initial transformations are crucial for being able to reduce the execution time down to a level in which further optimizations can be performed without sacrificing too much time for every test. Once the most basic techniques have been performed, vectorization also plays an important role in reducing the execution time. It is interesting to see that on average, the effect of this technique is negative (i.e. increments the execution time) in the *small* test case, but is beneficial in the *huge* test case. This emphasizes the importance of feeding the vectorial units with enough data, and also the importance of measuring the results with realistic test-cases, even when smaller ones are used for testing purposes. The effect of tiled and cache-oblivious techniques exhibits the opposite behavior; in this case, the number of extra added instructions to implement the tiled iterator and the recursion does not compensate the benefit in the memory management units for the huge testcase. It also complicates the automatic vectorization heuristics of the Intel compiler, making it more difficult to prove loop invariants and forcing the compiler to include run-time checks that can penalize the execution time. Even though parallelization (adding `#pragma` tags to certain loops) keeps being the most important optimization in terms of maximum improvement, the overall improvement cannot only be associated with having more cores available in the platform.

On the one hand, some transformations that do not increase performance "per se" (such as the loop transformation, which yields an average speed up close to 1, i.e. no significant improvement) are necessary for enabling other optimizations to be effective. This is because loop transformations enable the use of different optimizations in multiple kernels, instead of restricting their impact on only a single kernel.

On the other hand, the smart utilization of data structures such as double-buffering and vectorization also has a big impact on the overall improvement.

The same effect can be observed in the optimization of memory access patterns by means of tiled and cache-oblivious transformations of the source code. We can see in Fig. 12 that the mentioned optimizations only increase the average speed of the algorithm in the *small* test case, while they slow it down in the *huge* test case. As the main interest of the simulation is being able to obtain insightful results with the *huge* set of parameters, these optimizations have been disabled.

In the relative order of the optimizations, we can see the importance of data and instruction caches when estimating the performance of embedded code in high-performance computers. We show that in general, the effect of caches should not be neglected by performance estimation tools. Compilers should not neglect the importance of optimizing memory access patterns either. Although compilers do nowadays a great job in implementing automatically such optimizations, we have seen that in the majority of cases, because of the lack of context, compilers are not able to automatically infer the conditions that make these optimizations possible, and human assistance is usually needed. As a technical barrier that can be overpassed, the analysis of inter-

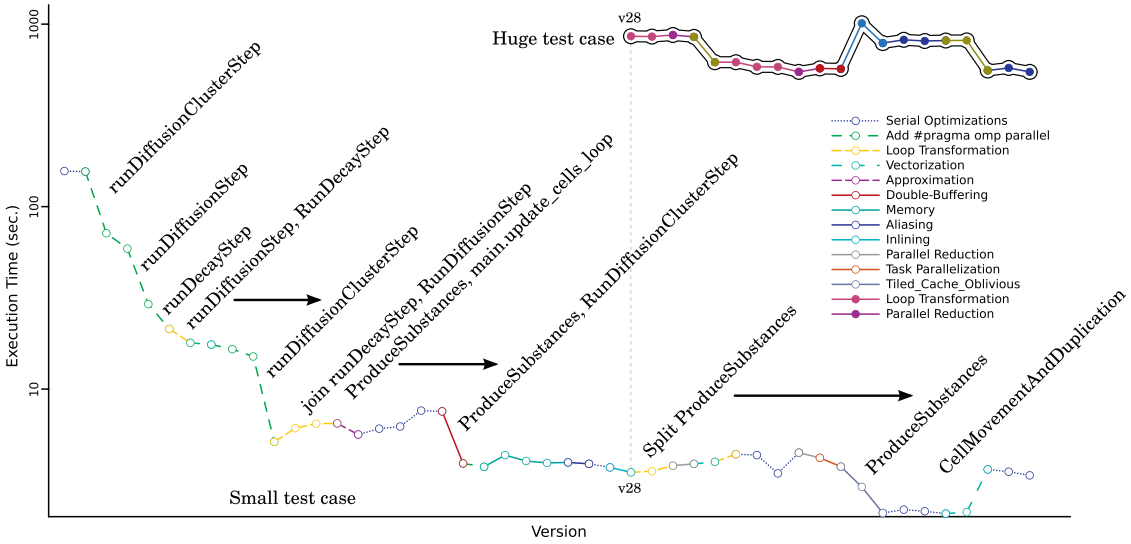


Fig. 11. Execution time for different versions ('huge' and 'small' test cases). Arrows indicate dependencies between specific optimizations (i.e. the effect of the pointed optimization is heavily influenced by having applied the previous transformation before). In the upper-right sector, the framed series of the graph correspond to the simulation times for the *huge* parameter set. Both series share the same *x*-axis (version), but due to time constraints (tasks that last more than 1000 s were killed, to ensure a fair cluster utilization among the participants), the *huge* test case was not analyzed before version 28.

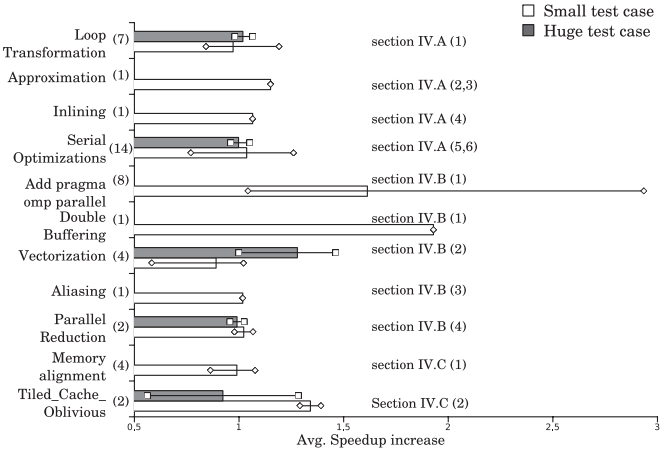


Fig. 12. Relative effect of different optimization techniques in the small and huge test cases. Note that first versions of the 'huge' test case cannot be simulated due to excessive execution time and lack of resources. Also note that some techniques can be applied more times than others, as indicated in parentheses. The bars show the average improvement of applying a given technique, as measured by computing the average improvement of the execution time after application of the technique. The error bars show the maximum and minimum value of all these improvements for a given technique. The large differences in the magnitudes of these error bars demonstrate that individual techniques can strongly vary in their impact, depending on the specific context of the code. The 'small' and 'huge' parameter sets are presented in Table 1. The locations of the explanations on the given techniques are indicated on the right-hand side.

procedural information is of key importance.

Finally, we emphasize that some optimizations require "losing" performance temporarily to be able to improve later on. Having an accurate intuition on how hardware and software interact will prevent us from "giving up" too soon in the optimization process and improve quickly later on. This is the case for example in loop transformations, that are required in order to expose more code to parallel execution, or loop splitting, to update different variables in different processors. In Fig. 11, these dependencies are presented with arrows.

5.1. Applicability to other platforms

In order to assess the applicability of the aforementioned techniques to other platforms, their effect on the execution time has been measured in new architectures different to the ones in which the competition took place, including several Intel Xeon and Xeon Phi processors and an AMD Opteron processor.

5.1.1. Intel architectures

The main architectural features of the tested Intel platforms are as follows:

- **Sandy Bridge:** Intel Xeon processor E5-2690 based on this architecture comprises 2 packages with 8 cores each, a total of 16 cores clocked at 2.90 GHz. The two packages form a 2-node NUMA (non-uniform memory access) system with a total of 64 GB of DDR3 memory at 1333 MHz. Each package contains 20 MB of level 3 (L3) cache, and additionally, each core is equipped with 256 kB of level 2 (L2) cache and 32 kB of level 1 (L1) data cache. The most modern vector instruction set supported by the processor is 256-bit AVX instructions.
- **Broadwell:** Intel Xeon processor E5-2699 v4 based on the Broadwell architecture comprises 2 packages with 22 cores each clocked 2.20 GHz. The 2-way NUMA system is built with 128 GB of DDR4 memory at 2400 MHz. There is 55 MB of L3 cache per package, and each core contains 256 kB of L2 cache and 32 kB of L1 cache. The processor supports 256-bit AVX and AVX2 instructions.
- **Knights Corner:** Intel Xeon Phi coprocessor 7120P is a PCIe add-in card with 61 cores at 1.24 GHz with symmetric access to 16 GB of GDDR5 memory at 2750 GHz. Each core contains 512 kB of L2 cache and 32 kB of L1 data cache. This architecture supports 512-bit Knights Corner instructions, also known as IMCI. The optimized code was executed using 4 threads per core (244 threads).
- **Knights Landing:** Intel Xeon Phi processor 7250 comprises 68 cores at 1.40 GHz (1.20 GHz AVX frequency). Cores have direct access to 96 GB of on-platform DDR4 memory at 1833 MHz and 16 GB of on-package MCDRAM in flat memory topology. The amount of L2 cache is 1 MB per tile of 2 cores and 32 kB of L1 data cache per core. The processor supports a 512-bit vector instruction sets called AVX-512 with AVX-512F, AVX-512CD, AVX-512ER and AVX-512PF modules.

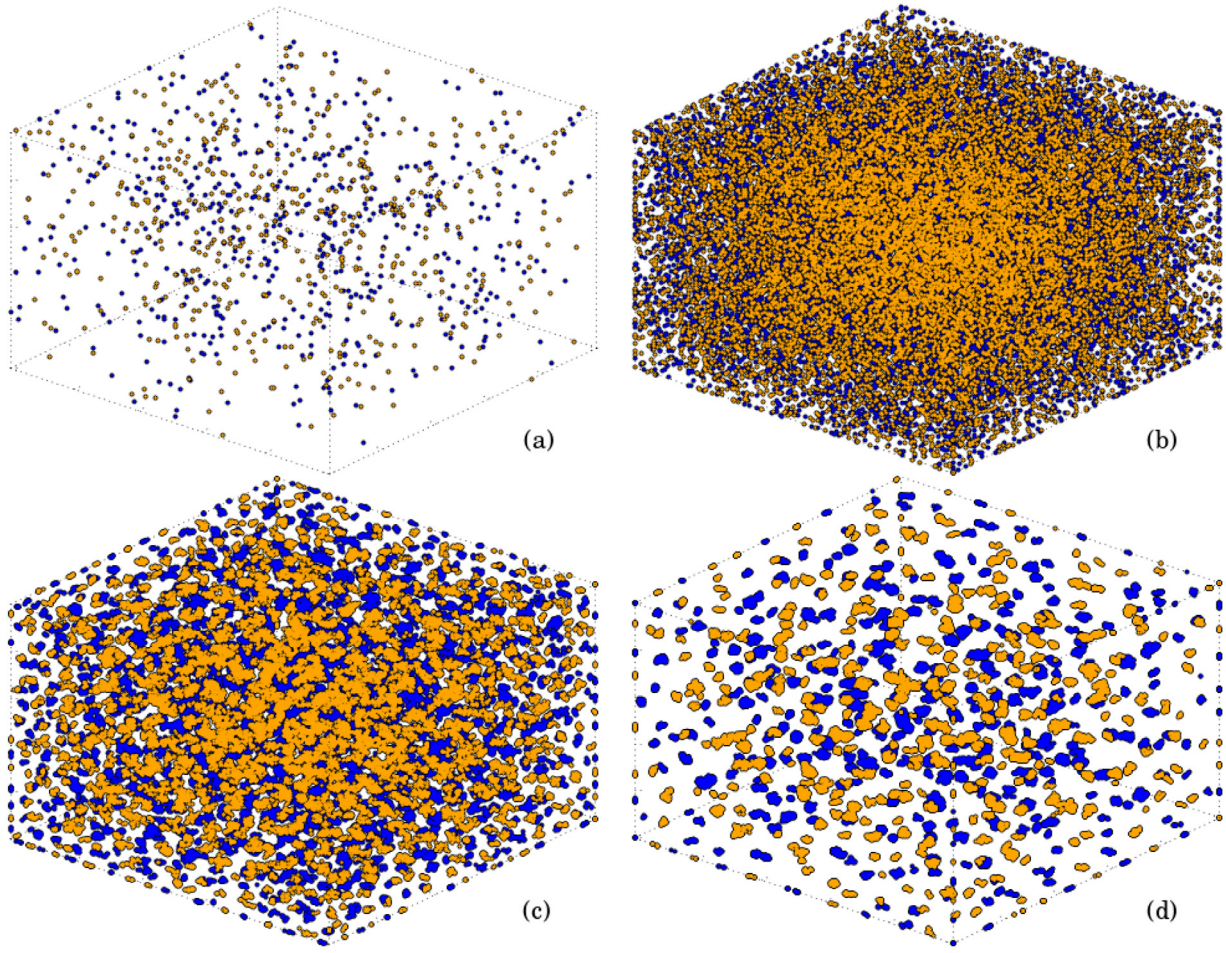


Fig. 13. Several steps of the simulation based on the 'huge' test case. In the top row, Phase 1 of the simulation is represented. In the bottom row, we can see the progressive clustering of cells. The overall simulation consists of 2^{25} cells, while only a subset of them is shown for clearer visualization. The simulation contains about the same number of cells as there are in one cm^3 of brain tissue. The biological time span of the simulated behaviors (cell proliferation, cell migration and pattern formation) ranges from several weeks to months, and so the simulation represents a biological time scale that is approximately 10,000 times longer than the overall simulation time.

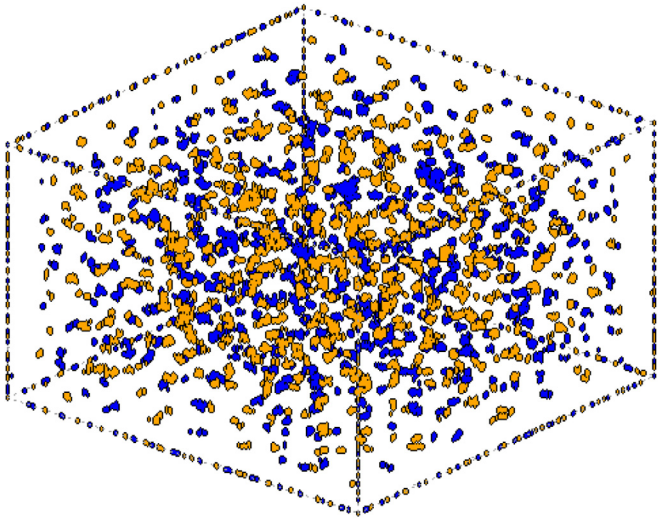


Fig. 14. Final distribution of cells in the non-clustering configuration given by the following parameters: speed = 0.01, $T = 500$, $L = 80$, $D = 0.0$, $\mu = 3.5$, $T_{Div} = 16$, $T_{Path} = 2.0 \Rightarrow$ final_energy = 2.907056e–05. The figure demonstrates that clustering arises only in specific parameter regimes, while remains partial in others.

Table 2 and Fig. 15 show the execution time of the optimized program and the "huge" set of parameters on the new platforms.

5.1.2. *Non-Intel platforms*

The code has also been tested on a 64-cores AMD Opteron 6376 machine, each core provided with 2048 kB of L1-cache and a globally connected RAM of 264 GB. In this platform the code has been compiled with gcc instead of icpc (Intel compiler). This implies several of the previously mentioned techniques are not available. In particular:

- The code is compiled without the "mmic" flag, so no particular optimizations regarding the Xeon-Phi architecture are applied.
- Since the Intel math-kernel-library specifically aims for performance gains on Intel platforms, we do not use it in this version of the code.
- Since gcc does not include the `__mm_malloc` function (that implements the first-touch policy previously explained), we do not use this policy. Instead, each memory is allocated in the common 264 GB RAM.
- Since `gcc` does not support array index notation, the source code is modified to remove the usage of this feature. `memset` function (from stdlib) is used instead.

The results of this new evaluation are shown in Figs. 15(a) and (b) and are discussed in the following section.

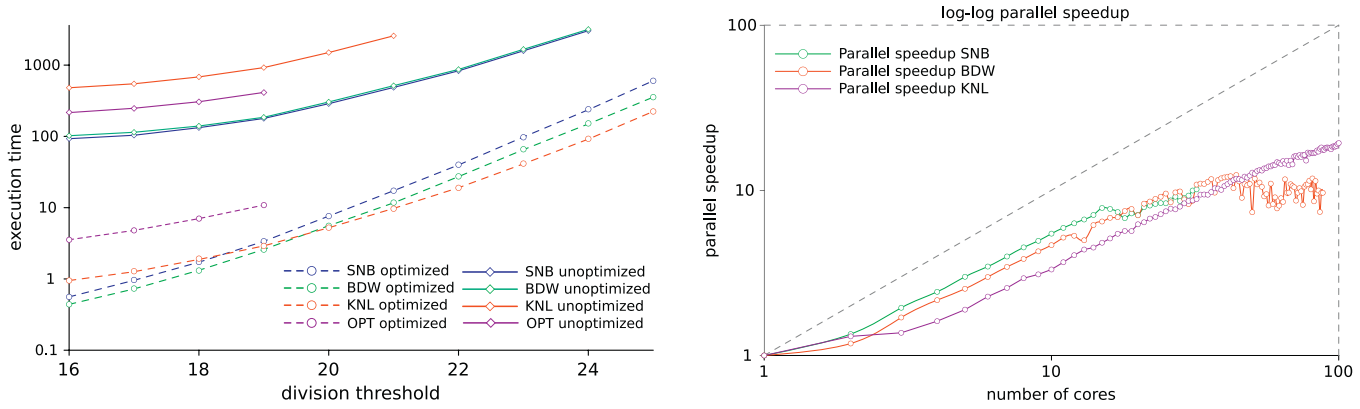


Fig. 15. Scalability (a) and speedup (b) of the original and optimized code in the three modern architectures described in Section 5.1. (a) shows the exponential nature of the computation when modifying the division threshold. The proposed optimizations enable large, multi-scale and agent-based simulations to become feasible. (b) presents the parallel speed-up (the speed-up of the proposed solution as a function of the number of cores, after taking into account all the sequential optimizations). Hence, the sequential performance improvement is not considered in (b). "SNB" refers to architecture "Sandy-Bridge", "BDW" to "Broadwell", "KNL" to "Knights-Landing" and "OPT" to "Opteron". Development platforms with "Knights-Corner" architecture were no longer available when these experiments were performed.

Table 1
Simulation parameters for 'small' and 'huge' test cases.

Small test case			Huge test case		
speed	=	0.01	speed	=	0.01
T	=	500	T	=	500
L	=	80	L	=	820
D	=	0.3	D	=	0.3
μ	=	0.1	μ	=	0.1
T_{Div}	=	16	T_{Div}	=	25
T_{Path}	=	2.0	T_{Path}	=	2.0

Table 2
Execution time (in seconds) of "small" and "huge" test cases with different platforms.

Computing platform	Small test case (non-optimized)	Small test case (optimized)	Huge test case (optimized)
Sandy Bridge	92.82	0.95	598
Broadwell	102.2	0.5	371
Knights Corner	155	2.08	491
Knights Landing	476	0.88	147
AMD Opteron	220	3.77	Not-enough-memory

6. Conclusion

In this paper, we studied and assessed modern coding techniques to demonstrate how simulation code of agent-based biological dynamics is optimized to increase performance, when run with many-core parallel processors. Our code simulates varying numbers of agents in 3D space, which interact and behave in many different ways; namely by proliferation, migration, secretion of diffusable substances, internal dynamics, and detection of external chemical gradients. Because of this diversity of interactions, the techniques presented in this paper can be extrapolated to many different contexts and problems. Studies that involve computer simulations based on some of these mechanisms range across a wide range of topics within the field of computational biology, such as for example the computational modeling of cancer progression [29], biofilm growth [30] or microbial system patterning [31]. Moreover, although this demonstration is in the context of biological systems, the same principles are applicable to a wide range of non-biological scenarios involving agent-based modeling.

We emphasize that the nature of the proliferation (i.e. cell division) imposes novel questions in terms of the optimized implementation, because it implies that the workload varies over time. Indeed, such dynamic system size is a common characteristic in many biological systems [32,33]. However, the code optimizations significantly improved performance despite such a dynamic demand in computing resources during simulation. Hence, our results demonstrate the applicability of our code improvements to simulations incorporating changing numbers of agents.

The optimizations described in this work have been driven both by the need to improve performance over a given machine that is already known and well established, but also with the need to be able to run the code in future generations of processors of the same family. As a consequence, all the optimizations are independent of the machine on which the code is executed. Intriguingly, the optimizations yield an even higher speed-up of 595 using the Intel Xeon Phi Knights Landing processors, even though the optimizations were implemented based on the Knights Corner platform (where the speed-up is 320-fold). This observation further highlights that our optimization approach is highly generalizable. Further investigation is being conducted to extend the obtained results to multiple computing nodes connected over a network. Distributed-memory parallel frameworks such as MPI are being used to this end [34–36].

The focus on high workload in the design of modern computational platforms and the importance of adapting the code to them can be observed in Fig. 15. We observe that a correct match between the architecture and a modern description of the code can lead to an improvement in the execution time of several orders of magnitude and enable the simulation of cases that were not feasible before. It is also worth observing the specificity of modern platforms, such as Knights Landing (KNL) for massive simulations. While we can see that KNL performed less favorable in the small test-case (division threshold = 16), the trend swaps when the size of the simulation transitions to a massive set of cells, where KNL outperforms the other architectures.

We discussed not only the main techniques to improve performance in the given code, but also the evaluation of correctness of the given implementations. The code to be optimized is genuine in the sense that it serves as the source for future high-performance, agent-based simulations of biological dynamics, so the correctness of the parallel implementation is of great importance. Although the concrete relative effect of each optimization technique might vary, we present a common language and approach for describing optimizations and the analysis of

its relative importance, serving as a guidance for other projects.

Overall, the optimizations have produced several hundred-fold shorter execution times, reducing the order of time from days to minutes. These results demonstrate excellent parallel scalability and general applicability to various platforms. Finally, they show that current optimization techniques allow for dynamic and efficient usage of computational resources, which is crucial for simulations of many agent-based models in computational biology.

Authors' contributions

P.G. conducted the code optimization, the computer simulations shown in the paper and wrote the paper. M.M. and M.K. aided in the design of the manuscript. J.B. contributed to the organization of the Intel Modern Code Developer Challenge. A.V. and R.A. configured the computing cluster and contributed to the preparation of the hardware-related part of the manuscript. R.B. implemented the sequential base code, the evaluation criterion used to assess the correctness of the submissions, and contributed to the preparation of the manuscript.

Acknowledgments

The authors would like to thank Fons Rakemakers for valuable and interesting discussions on the topic, and Intel for the Intel Modern Code Developer Challenge 2015. In particular, Jason Sewall for edits to the initial code that improved the overall code structure and readability.

P.G. gratefully thanks the funding and support from the Engineering and Physical Sciences Research Council (EPSRC) under grant PAMELA EP/K008730/1. R.B. and M.K. were supported by the Human Green Brain Project (http://www.greenbrainproject.org) through the EPSRC (EP/K026992/1). R.B. was also supported by the Medical Research Council of the UK (MR/N015037/1). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

References

[1] Izhikevich EM, Edelman GM. Large-scale model of mammalian thalamocortical systems. Proc Natl Acad Sci 2008;105(9):3593–8.
[2] Potjans W, Morrison A, Diesmann M. Enabling functional neural circuit simulations with distributed computing of neuromodulated plasticity. Spike-Timing Dependent Plasticity. 2010. p. 357.
[3] Karr JR, Sanghvi JC, Macklin DN, Gutschow MV, Jacobs JM, Bolival B, Assad-Garcia N, Glass JI, Covert MW. A whole-cell computational model predicts phenotype from genotype. Cell 2012;150(2):389–401.
[4] Bauer R, Zubler F, Pfister S, Hauri A, Pfeiffer M, Muir DR, Douglas RJ. Developmental self-construction and-configuration of functional neocortical neuronal networks. PLoS Comput Biol 2014;10(12):e1003994.
[5] Freund JB. Numerical simulation of flowing blood cells. Annu Rev Fluid Mech 2014;46:67–95.
[6] Furber SB, Galluppi F, Temple S, Plana LA. The spinnaker project. Proc IEEE 2014;102(5):652–65.
[7] Tomsett RJ, Ainsworth M, Thiele A, Sanayei M, Chen X, Gieselmann MA, Whittington MA, Cunningham MO, Kaiser M. Virtual electrode recording tool for extracellular potentials (vertex): comparing multi-electrode recordings from simulated and biological mammalian cortical tissue. Brain Struct Funct 2015;220(4):2333–53.
[8] Farmer JD, Foley D. The economy needs agent-based modelling. Nature 2009;460(7256):685–6.
[9] Deissenberg C, Hoog SVD, Dawid H. Eurace: a massively parallel agent-based model of the european economy. Appl Math Comput 2008;204(2):541–52.
[10] Harvey DG, Fletcher AG, Osborne JM, Pitt-Francis J. A parallel implementation of an off-lattice individual-based model of multicellular populations. Comput Phys Commun 2015;192:130–7.
[11] Fachada N, Lopes VV, Martins RC, Rosa AC. Parallelization strategies for spatial agent-based models. Int J Parallel Prog 2017;45(3):449–81.
[12] Kang G, Márquez C, Barat A, Byrne AT, Prehn JHM, Sorribes J. Eduardo César colorectal tumour simulation using agent based modelling and high performance computing. Fut Gener Comput Syst 2017;67:397–408.
[13] Chen B, Kantowski R, Dai X, Baron E, Van der Mark P. Accelerating gravitational microlensing simulations using the Xeon Phi coprocessor. Astron Comput 2017;19:60–5.
[14] Parks C, Huang L, Wang Y, Doraiswami R. Accelerating multiple replica molecular dynamics simulations using the intel® Xeon Phi coprocessor. Mol Simul 2017;42(9):714–23.
[15] Insel TR. Rethinking schizophrenia. Nature 2010;468(7321):187–93.
[16] Hagerman RJ. Epilepsy drives autism in neurodevelopmental disorders. Dev Med Child Neurol 2013;55(2):101–2.
[17] Hu WF, Chahrour MH, Walsh CA. The diverse genetic landscape of neurodevelopmental disorders. Annu Rev Genom Hum Genet 2014;15:195–213.
[18] Bauer R, Breitwieser L, Meglio AD, Johard L, Kaiser M, Manca M, Mazzara M, Rademakers F, Talanov M, Tchitchigin AD, Global I. The biodynamo project: experience report. Advanced Research on Biologically Inspired Cognitive Architectures. 2017. p. 117–25.
[19] Vea D. https://www.software.intel.com/en-us/articles/case-study-optimized-code-for-neural-cell-simulations. Last time accessed: 2017-10-11.
[20] Jin T. Gradient sensing during chemotaxis. Curr Opin Cell Biol 2013;25(5):532–7.
[21] Bauer R, Zubler F, Hauri A, Muir DR, Douglas RJ. Developmental origin of patchy axonal connectivity in the neocortex: a computational model. Cerebral Cortex 2012;24(2):487–500.
[22] Vladimirov A, Asai R, Karpusenko V. Parallel Programming and Optimization with Intel Xeon Phi Coprocessors. 2nd ed. Colfax International; 2015. ISBN 978-0-9885234-0-1
[23] Little SC, Tikhonov M, Gregor T. Precise developmental gene expression arises from globally stochastic transcriptional activity. Cell 2013;154(4):789–800.
[24] McCool MD, Robison AD, Reinders J. Structured Parallel Programming: Patterns for Efficient Computation. Elsevier; 2012. ISO 690
[25] Zhang S, Zhang T, Wu Y, Yi Y. Flow simulation and visualization in a three-dimensional shipping information system. Adv Eng Softw 2016;96:29–37.
[26] Borin E, Devloo PR, Vieira GS, Shauer N. Accelerating engineering software on modern multi-core processors. Adv Eng Softw 2015;84:77–84.
[27] Cai Y, Wang G, Li G, Wang H. A high performance crashworthiness simulation system based on GPU. Adv Eng Softw 2015;86:29–38.
[28] Velivelli AC, Bryden KM. Domain decomposition based coupling between the lattice Boltzmann method and traditional CFD methods. Part I: formulation and application to the 2-D burgers equation. Adv Eng Softw 2014;70:104–12.
[29] Macklin P, Edgerton ME, Thompson AM, Cristini V. Patient-calibrated agent-based modelling of ductal carcinoma in situ (dcis): from microscopic measurements to macroscopic predictions of clinical progression. J Theor Biol 2012;301:122–40.
[30] Popławski NJ, Shirinifard A, Swat M, Glazier JA. Simulation of single-species bacterial-biofilm growth using the Glazier-Graner-Hogeweg model and the CompuCell3D modeling environment. Math Biosci Eng 2008;5(2):355.
[31] Kang S, Kahan S, Momeni B. Simulating microbial community patterning using biocellion. Engineering and Analyzing Multicellular Systems: Methods and Protocols. 2014. p. 233–53.
[32] Zubler F, Hauri A, Pfister S, Bauer R, Anderson JC, Whatley AM, Douglas RJ. Simulating cortical development as a self constructing process: a novel multi-scale approach combining molecular and physical aspects. PLoS Comput Biol 2013;9(8):e1003173.
[33] Bauer R, Kaiser M. Nonlinear growth: an origin of hub organization in complex networks. R Soc Open Sci 2017;4(3):160691.
[34] de la Asuncin M, Castro MJ, Mantas JM, Ortega S. Numerical simulation of Tsunamis generated by landslides on multiple GPUs. Adv Eng Softw 2016;99:59–72.
[35] Nikoli M, Hajdukovi M, Milainovi DD, Gole D, Mari P, ivanov. Hybrid MPI/openMP cloud parallelization of harmonic coupled finite strip method applied on reinforced concrete prismatic shell structure. Adv Eng Softw 2015;84:55–67.
[36] Milainovi DD, Borkovi AI, Raki PS, Nikoli M, Strievi L, Hajdukovi M. Large displacement stability analysis of thin plate structures: scope of MPI/openMP parallelization in harmonic coupled finite strip analysis. Adv Eng Softw 2013;66:40–51.